A Go-to-FPGA Compilation Framework for Streaming Applications

Tianyi Cui
Advisor: Prof. Jason Cong

November 9, 2017

Abstract

In this report, we show the potential advantage of Go programming language that leverages communicating sequential processes (CSP) in FPGA accelerator development. We build a compilation framework to compile a Go program to an FPGA accelerator. The compiler covers two optimizations, fine-grained parallelism and take-level pipeline, that we identified based on the CSP to improve the throughput of generated accelerator designs. Finally, we analyze in this report that several applications could benefit from the proposed framework.

1 Introduction

Due to the end of Moore's law, conventional general-propose processor can no longer obtain high performance with efficient energy consumption. Heterogeneous computer architecture has been widely used as a new method to improve the performance while maintain the energy efficiency. In the heterogeneous computing architecture, field programmable gate arrays (FPGA) plays an important role because of its energy efficiency, high reconfigurability and capability of parallelism.

The programmability for FPGA is still an open research problem. Traditional method for developers to implement applications on FPGA is based on hardware description language (HDL), which requires developers to be experts in digital circuits. Specifically, developers are required to be ware of all FPGA details such as PCIE-protocols, on-chip memory organization, pipelining and so forth. This poor programmability largely obstructs developers to use FPGA as accelerators in heterogeneous computing platform.

Fortunately, the appearance of high-level synthesis (HLS)[2] enables software developers to program FPGA easily. Developers are allowed to design FPGA using high-level languages like C/C++. However, C/C++ is a sequential programming language which might not easy for the compiler to recognize the optimal scheduling. As a result, previous work provides a set of pragmas to guide the compiler for improving the performance of generated designs [4][1]. These solutions, however, still require software developers to have background knowledge about FPGA for developing efficient accelerators.

In order to reduce the impediment for software programmers to program FPGAs, Communicating sequential processes (CSP) can potentially be a suitable candidate. Communicating sequential processes (CSP) is the concept to describe patterns of interaction in concurrent systems described in 1977. It leverages channels to conduct message passing between different entities. This kind of feature is especially suitable for FPGA since it could express parallelism instinctively. To leverage the power of CSP, in this project, we design and implement a compilation framework that leverages Go programming language as the input and generates efficient FPGA accelerators. Go programming language (Golang) is increasingly popular among all the programming languages in recent years. Unlike C language, Golang takes the advantage of CSP programming model and leverages coroutines and channels to express parallelism. Thus, it is more nature for software programmers to explicitly express the parallelism and for compiler to generate efficient hardware accelerator. This project makes

the following contributions:

- We build a framework to compile out-of-box Go programs to FPGA.
- We propose two optimizations to improve the performance.
- We study several applications to illustrate the benefit of designing FPGA accelerators using Golang.

In the rest of this report, we will first mention several useful features in Go programming language in section 2. We then discuss our overall design in section 3. Based on the proposed design, we will summarize two challenges for implementing the compiler to generate efficient hardware in section 4. We mention our optimizations to address the challenge in section 5 Finally, we will state our current progress and future work in section 7.

2 Golang Features

Go is an expressive, concise, clean, and efficient programming language. Its three features are especially useful for FPGA development: Go routine, channel and closure.

2.1 Go routine

Go routine is lightweight thread. Developers could use "go f()" to call a function "f" to create a go routine. Figure 1 shows an example of Go routine. In this example, in the main function, the program accept requests continuously and process each request in parallel.

```
func deal_conn(conn *Tcpconn) {
    //some code to process
connection
}
func main() {
    for {
        conn := in.accept()
        go deal_conn(conn)
    }
}
```

Figure 1: Go routine example

In FPGA development, Go routine can be used to express parallelism to explicit indicate compiler which part of the program could run in parallel. As a result, compiler could generate each Go routine and let them run in parallel to fully utilize parallelism in FPGA.

2.2 Channel

Channels are pipes that connect concurrent Go routines. In FPGA development, we could use channel to connect Go routines to express the data dependency between different Go routines. One example is shown in Figure 2.

```
func calc(addr_in chan,data chan)
{
  for
    addr := <-addr_in
    if (ishit(addr)) {
    //fast path
      data <- buffer[addr]
    } else { //slow path
      chan2m := make(chan)
      chan2m <- addr
      go fetch_data(chan2m,data)
    }
}
```

Figure 2: channel example

In this example, by using go routine and channel, we could offload slow path in the if condition clause to improve throughput. This example illustrate an application which requires to fetch data from DRAM. If the data is already in the buffer, then the if condition goes to fast path, else goes to the slow path. By leverage channels and go routines, we could offload the slow path to a different kernel to improve throughput by reducing the time wasted by waiting for slow path.

2.3 Closure

In programming languages, lexical closures are techniques for implementing lexically scoped name binding in languages with first-class functions. Operationally, a closure is a record storing a function together with an environment: a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the

name was bound when the closure was created. A closure?unlike a plain function?allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope. [6]

In FPGA design, closure could be used to express data dependency. By using a variable defined out of the function, compiler could know that there is a data dependency between the function call and the variable. After knowing the data dependency, compiler could generate efficient hardware more easily.

```
func AES_CBC() func() int {
  state := 0
  return func(blk int) int {
    state,result:=calc(blk,state)
    return result
    }
}
```

Figure 3: closure example

Figure 3 is an example. function AE_CBC returns a function which has the dependency of variable state.

3 Overall Design

Our Golang to FPGA compilation framework is shown in figure 4. The Go programs inputed by the developers are parsed by the frontend of the compiler. Our frontend parses the Go program to AST and generate corresponding HLS C code. After that, the code is sent into the optimizer to conduct backend optimizations. Finally, the generated HLS C code is synthesized by Vivado HLS toolchain.

3.1 Frontend

In the frontend part, we first parse the Go code to AST by take advantage of the parser of Go. We traverse the Abstract Syntax Tree to conduct source to source code generation.

We adopt Xilinx HLS C dataflow mode to generate Go routines and channels. Each Go routine is synthesized to one subkernel in HLS C and channels are converted to HLS stream in Xilinx HLS

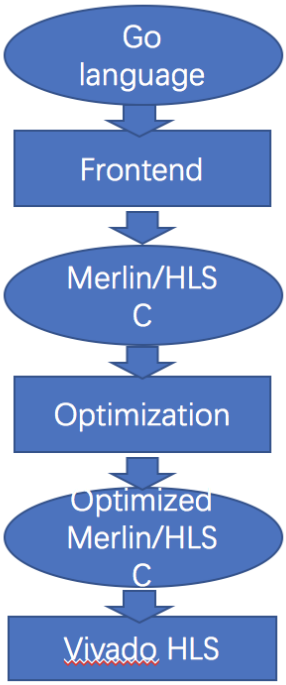

Figure 4: Golang compiler framework

Go syntax	HLS C syntax
Goroutine	subkernel
channels	hls::stream

Table 1: Syntax transformation for Go.

C. Table 1 shows the code transformation in the frontend part.

3.2 Backend

The backend part of the framework is to conduct optimizations. We could leverage Go routines and channels in Go programming language to generate dataflow graph. As stated in previously, we could adopt HLS C or Merlin Compiler to conduct optimizations within each block in the dataflow graph. The key focus of the backend is the dataflow level optimizations.

4 Challenges

Instinct code transformations mentioned above cannot generate efficient FPGA hardware. During our practice, there are two challenges remain unsolved after conduct code transformation.

4.1 Low Resource Utilization

After directly mapping each Go routine to a sub-kernel in FPGA. We notice that the utilization rate of the most critical resources (LUT or DSP) is usually not high enough, like about 50%. This lead to low throughput. However, we cannot simply duplicate the full dataflow graph because if the resource utilization will be almost 100% and will lead to the failure of synthesis.

4.2 Strong Data Dependency

In some applications (SHA-1, AES_CBC), the input of the same task is divided to blocks. The result of the current block is depends on the result of the previous block. However, we could get a very long pipeline with low initial interval to calculate the result of one block. Thus, if we use the instinct ways to generate HLS code, the result will be the pipeline is under-utilization since the next block cannot be calculated until the previous one is finished in the same task.

5 Optimizations

To tackle the two challenges mentioned above. We need to conduct optimizations in the backend of our compiler. To fully utilize resources on FPGA ,we leverage fine-grained parallelism. To tackle the strong dependency challenge, we leverage task-level pipeline.

5.1 Fine-Grained Parallelism

With our observation, we could notice that lots of applications will call the same Go routine for many times. Since we cannot simply duplicate the full datagraph. We could only duplicate a small portion of the program. By doing so, we reduce the number of kernels which are called for many times. Instead, we build scheduler to schedule the task to the same kernel. By doing so, we could reduce the resource cost and get reasonable speedup.

More specifically, this optimization has two situations:

5.1.1 Task without Data Dependency

For this case, the scheduler could be simply designed as round-robin, as shown in Figure 5 and 6.

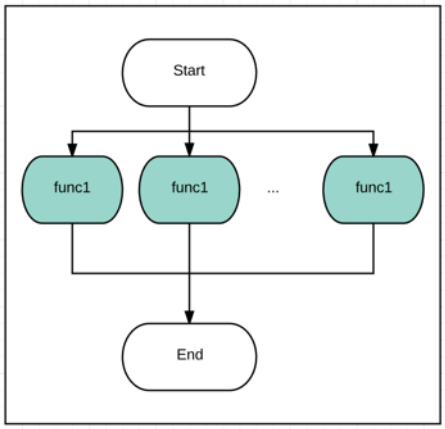

Figure 5: Original hardware design

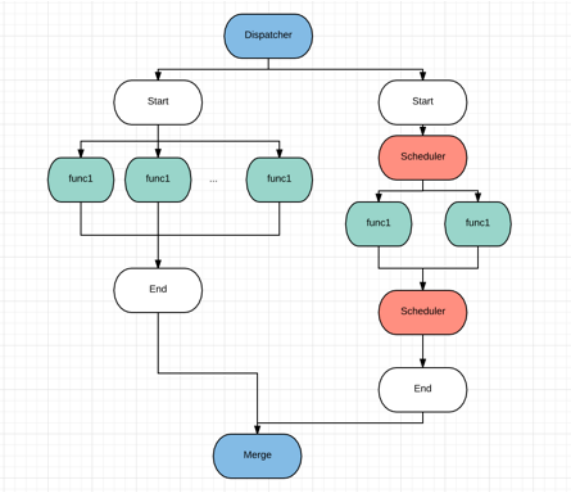

Figure 6: Optimized design

Figure 5 shows the original design on FPGA, which needs to duplicate "func1" for many times. However, we cannot duplicate the whole original

design due to the limitation of the resources. Thus, we need to only duplicate a limited number of func1 and build a scheduler to round-robin schedule the call of func1 as shown in Figure 6.

5.1.2 Task with Data dependency

In this case we need to maintain state of each task in the scheduler, as shown in 7 and 8.

Figure 7: Original design

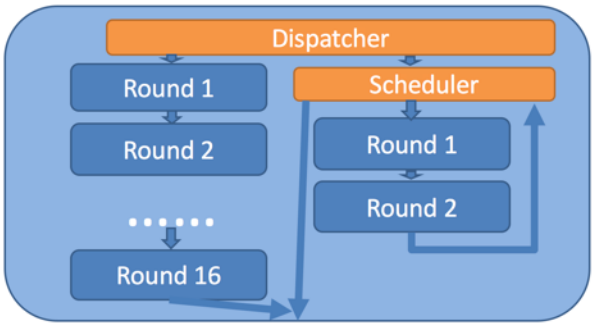

Figure 8: Optimized design

5.2 Task-Level Pipeline

We leverage Task-level pipeline to tackle the problem of strong dependency.

By doing so, we could put the second task into the PE when the first task is still running. As a result, the pipeline will not be blocked anymore.

The key challenge in this problem is how to identify the dependency and infer the task-level pipeline systematically and automatically. Our solution is to take advantage of the function closure of the Go programming language. Specifically, we could implement the code shown in figure 9:

```
for (each flow as f){
  times,enc_f:=aes(blocks in flow)
  go func(){
    for i:=0;i < times;++i{
      result[f][i]:=enc_f()
    }
  }
}
```

Figure 9: Code to express dependency

The second line gets the function closure by calling "aes" function. By using function closure, we could realize how the data dependency is involved in the program. Thus, we could infer the architecture as Figure 10:

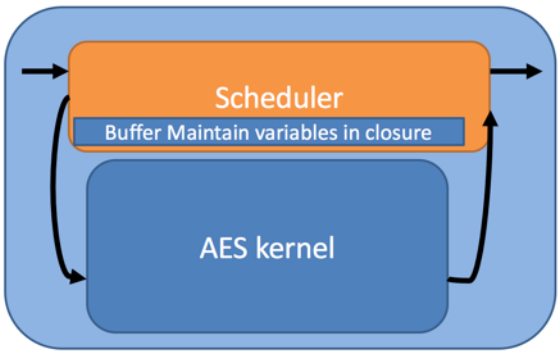

Figure 10: Optimized design after adopting task-level pipeline

By leverage task-level parallelism, the performance of AES_CBC could be improved by 70x.

6 Discussion

In this section, we choose three applications to show how optimizations mentioned in section 5 could be used in real applications.

6.1 equalizer

Equalization or equalisation is the process of adjusting the balance between frequency components within an electronic signal.[7]. The dataflow graph

Recources	BRAM_18k	DSP48E	FF	LUT
Total	10	654	88994	89593
Available	2940	3600	866400	433200

Table 2: Resources utilization of Equalizer

is shown in Figure 11. The data is duplicated and is transfered to each node in the graph. Developers could use Go routines and channels to express this dataflow graph easily.

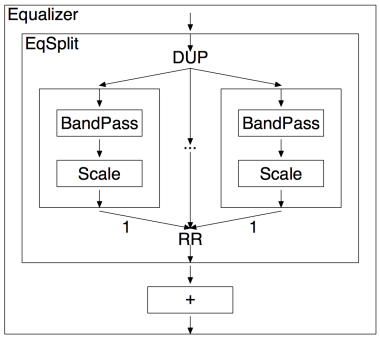

Figure 2: Stream graph for an equalizer

Figure 11: Equalizer

Our compiler could generate optimized design with initial interval(II)=1 with 250MHz frequency on Xilinx ADM-PCIE-7V3 board, the throughput achieves 225.4Mops. Table 2 shows the detailed resources utilization.

This design can be scaled up by duplicate the whole design until it uses up all the resources available.

6.2 DES

The Data Encryption Standard (DES) is a symmetric-key algorithm for the encryption of electronic data.[5]. Figure 12 shows the process of DES algorithm. It uses the same function for 16 times to get the cypher text of a block.

Currently, the naive version of our compiler could achieve the performance of Initial Interval(II)=1 with frequency of 270MHz on Xilinx ADM-PCIE-7V3 board, the throughput achieves 2.1GBps. Table 3 shows resources utilization of the design.

Since the resources are not fully utilized, our naive design does not achieve optimal. However, we

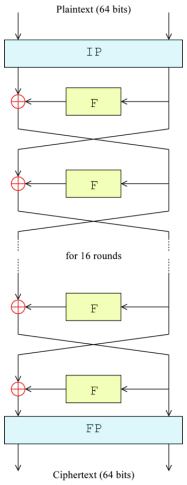

Figure 12: DES Algorithm (From Wikipedia)

Recources	BRAM_18k	DSP48E	FF	LUT
Total	80	0	63928	211602
Available	2940	3600	866400	433200

Table 3: Resources utilization of DES case

cannot simply duplicate the whole design since the utilization rate of LUT will grow up to almost 100% and lead to the failure of bitstream generation. The best choice for this design is to only duplicate some part of the design and build a scheduler to schedule the task. So we could adopt fine-grained parallelism mentioned above to get the optimized design for FPGA.

6.3 AES Cipher Block Chaining

The Advanced Encryption Standard (AES) is a kind of symmetric-key encryption algorithm which is widely used in security fields [3].

Cipher Block Chaining (CBC) is widely used to prevent frequency analysis. As shown in figure 13, the result of current block depends on the result of previous block, which leads to strong data dependency. On the other hand, block cipher encryption algorithm takes 70 cycles to get the result of a block and could be easily pipelined.

In this case, if we adopt intuitive design in HLS C. The result is that initial interval (II)=70 for the design. As a result, pipeline is not fully utilized in the design. However, we could easily adopt task

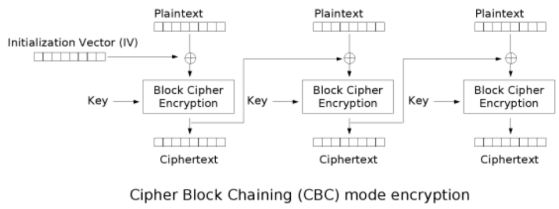

Figure 13: Cipher Block Chaining (From Wikipedia)

level pipeline to fully utilize the resources, in which II could. In this case, the throughput could be improved by 70x after adopting task level pipeline optimization.

7 Current Status and Future Plan

Currently, we have already built a preliminary frontend of the compiler, which support Go routine and channel syntax in Go programming model. We have already implemented a naive version of DES algorithm.

In the future, I will finish the backend optimizations mentioned above and finish all the syntaxes in Go programming language.

References

[1] Merlin compiler. `https://www.xilinx.com/products/intellectual-property/1-dmwshy.html`. Accessed: 19-September-2017.

[2] Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., and Zhang, Z. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 30*, 4 (April 2011), 473–491.

[3] Daemen, J., and Rijmen, V. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[4] Feist, T. Vivado design suite. *White Paper 5* (2012).

[5] Standard, D. E. Data encryption standard. *Federal Information Processing Standards Publication* (1999).

[6] Wikipedia. Closure (computer programming) — wikipedia, the free encyclopedia, 2017. [Online; accessed 19-September-2017].

[7] Wikipedia. Equalization (audio) — wikipedia, the free encyclopedia, 2017. [Online; accessed 20-September-2017].